Angular

Les stores avec NGRX

Expliqué par David Gilson

https://github.com/gilsdav

Assets

- https://github.com/gilsdav/aa-ngrx-workshop

Sommaire

- Qu’est-ce qu’un store ?
- Pourquoi utiliser un store ?
- Architecture Redux
  - Actions
  - Reducers
- Implémentation avec NGRX
- Gestion des side effects

1) Qu’est-ce qu’un store ?

Qu’est-ce qu’un état ?

- Toutes données non statique qui varient selon l’utilisation de l’application
- Exemple:
  - La réponse d’un service web
  - Le token d’authentification
  - Des données entrées par un utilisateur (filtre/recherche)
  - Gestion de l’interface (langue/messages)
  - Gestion de l’historique des routes
  - Et tout le reste...

Qu’est-ce qu’un store ?

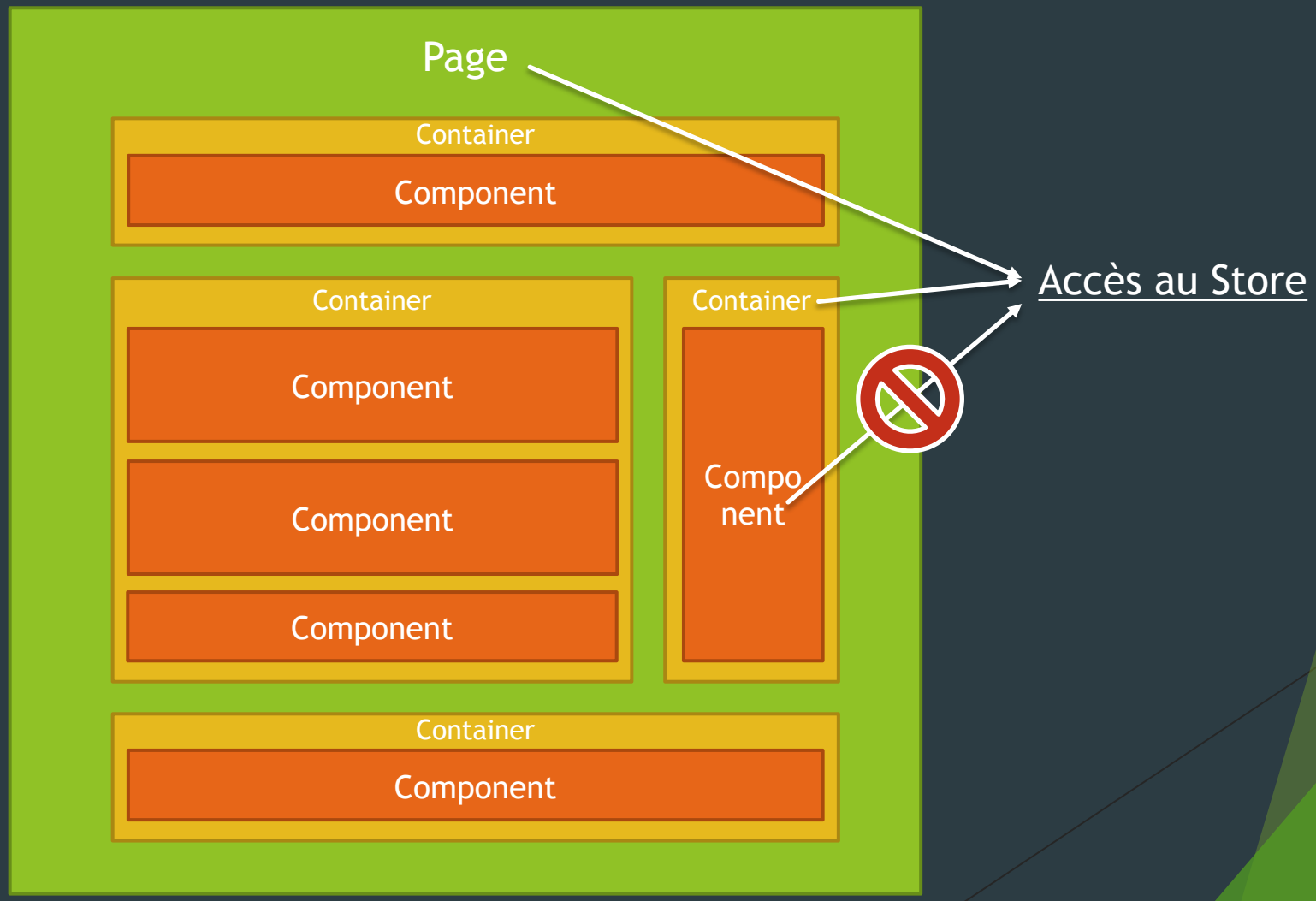
- Le store est un gestionnaire d’état (state)
  - Il initialise/modélise un état de base
  - Il connait et peut donner son état à un instant T
  - Il peut modifier son état à la demande (lui et lui seul)
  - Il monitor et observe les changements d’état

2) Pourquoi utiliser un store ?

Pourquoi utiliser un store ?

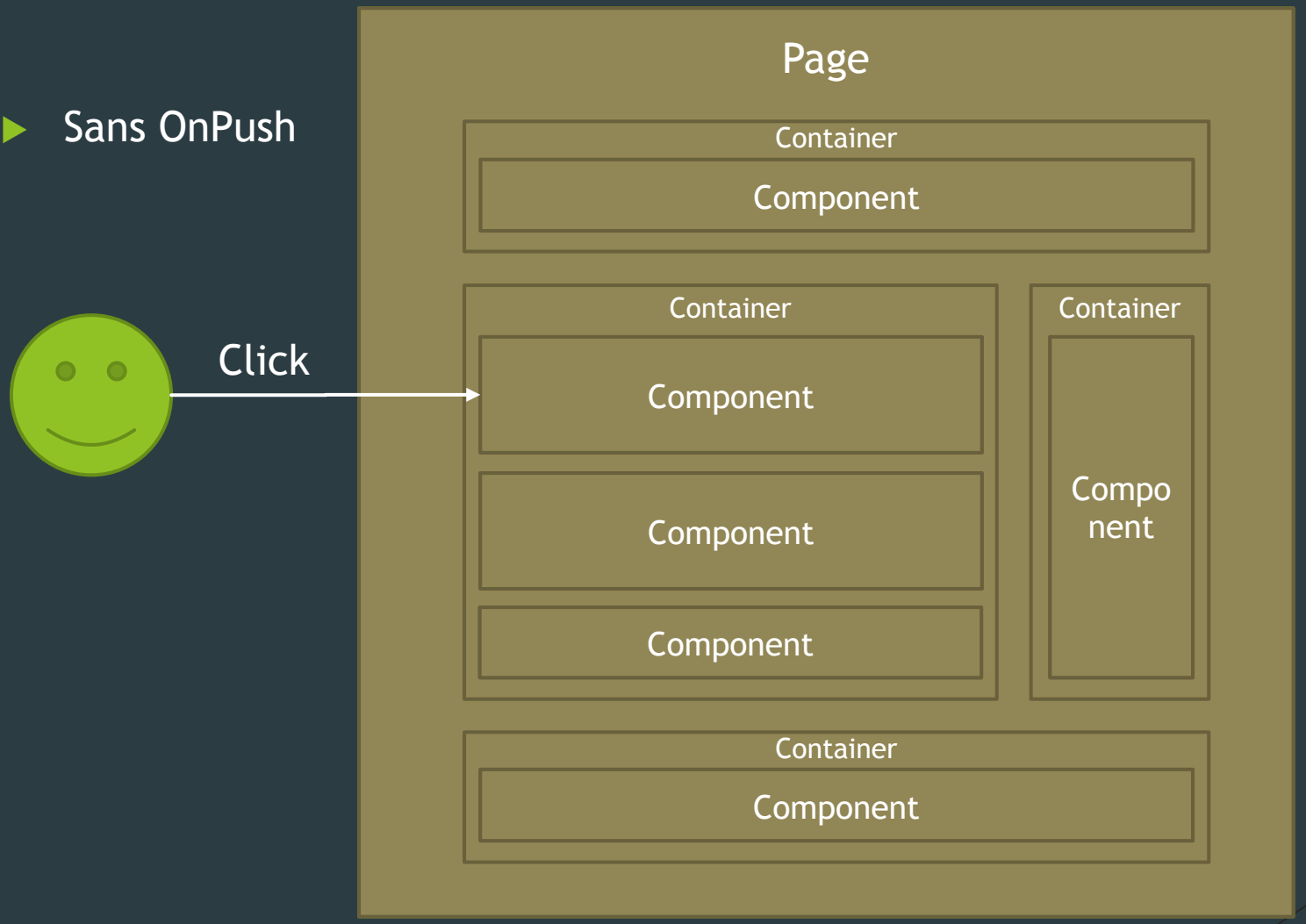
- Unique source de vérité
- Facilite de testing
- Amélioration des performances
  - ChangeDetection.OnPush
  - @Input immutables
    - La vérification par référence est extrêmement rapide par rapport à une vérification en profondeur
- Réduit le nombre d'Input/Output

Structure d’une application Angular



Fonctionnement du OnPush

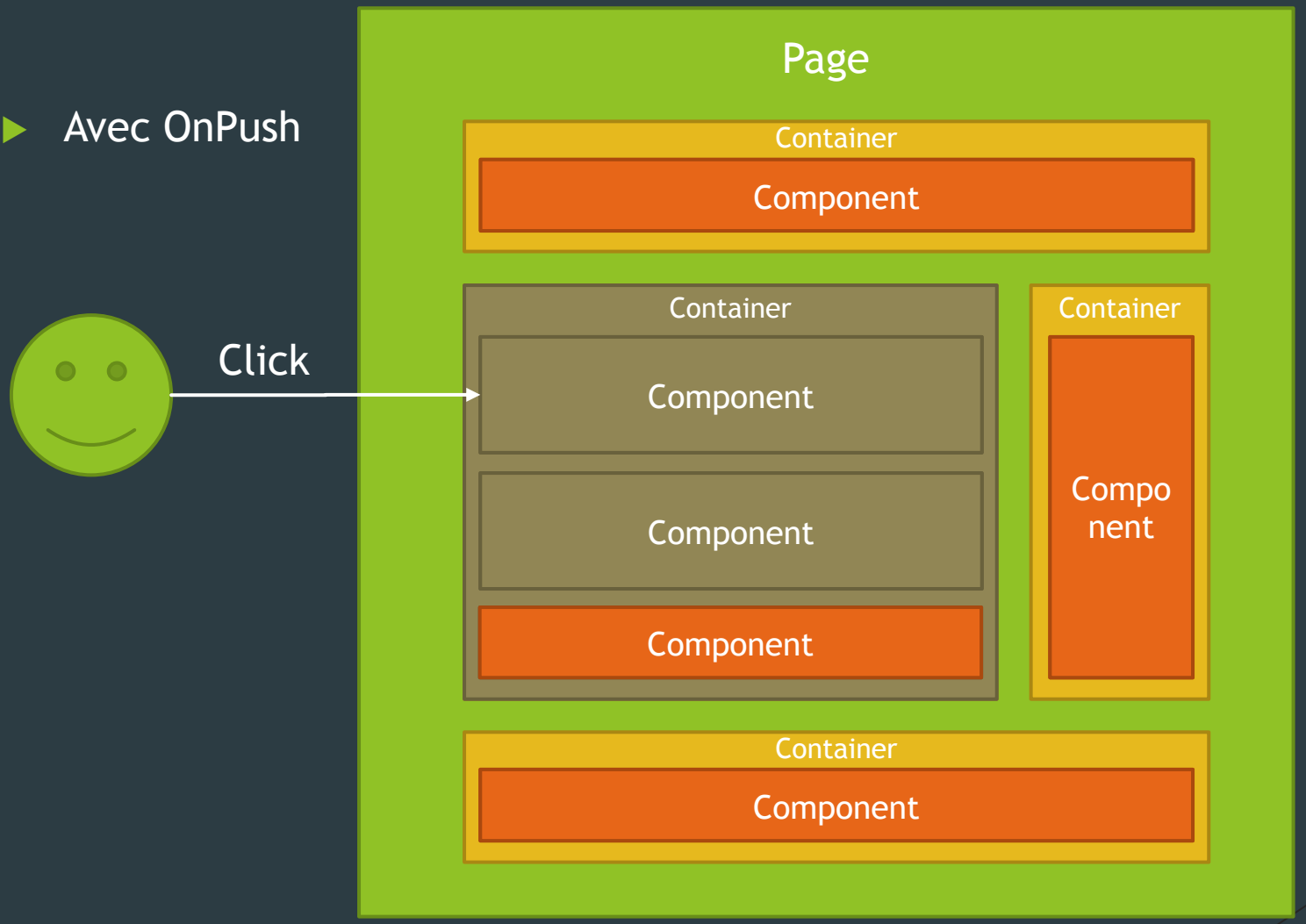
- Sans OnPush



Changement de couleur = nouveau rendu

Fonctionnement du OnPush

- Avec OnPush



Changement de couleur = nouveau rendu

Fonctionnement du OnPush

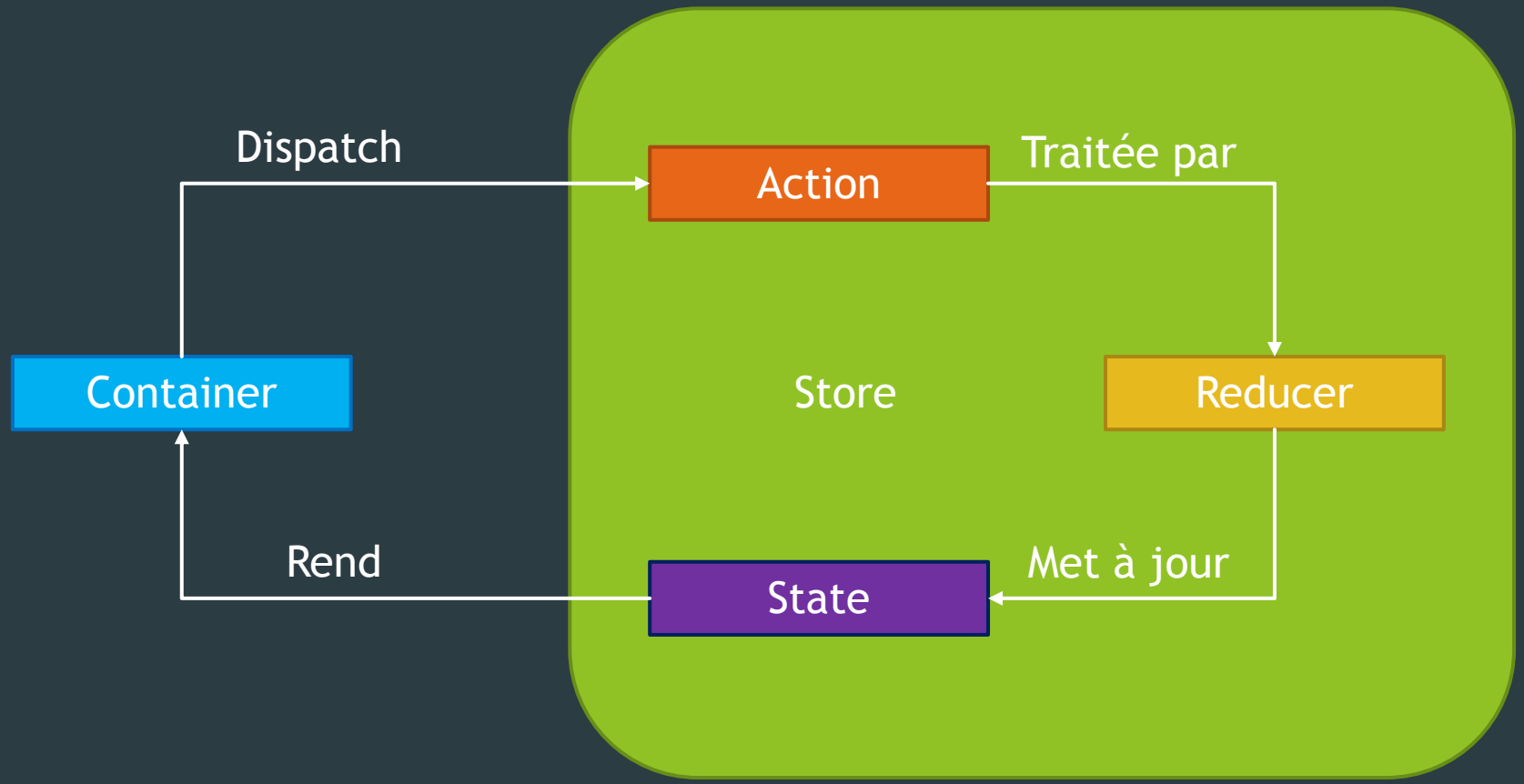
- Ce qui est détecté par un composant OnPush
  - Un changement synchrone suite à un évènement émit par le client (exemple: click) sur le composant
  - Le changement de valeur d'un Observable qui est souscrit dans le composant
  - Le changement de valeur d'un @Input du composant

- Doit-on mettre tous ces « subscribe » dans le ngOnInit ?
  - Si la valeur n'est utilisée que dans la partie HTML (exemple: passer l'information d'un container à un composant visuel), on utilise le pipe « **async** » qui va souscrire et dé-souscrire à notre place pour récupérer la valeur de l'observable.

```
<myComponent [value]="value$ | async"></ myComponent>
```

3) Architecture Redux

Architecture Redux



Architecture Redux - un arbre d'états unique

- Simple objet JavaScript
- Géré par un ou plusieurs reducers

```
const state = {
    todos: []
}
```

Architecture Redux - actions

- Objet JavaScript
- Deux propriétés
  - type: string qui représente la demande
  - payload: les données nécessaires pour le traitement de la demande (optionnel)

```
const action = {
    type: 'ADD_TODO',
    payload: {
        label: 'Eat pizza',
        complete: false
    }
}
```

Architecture Redux - reducers

- Fonction pure
- Deux paramètres
  - state: l'état précédent
  - action: demande à traiter
- Utilise le payload selon le type de l'action pour générer un nouvel état
- Retourne le nouvel état

Architecture Redux - reducers

```
function reducer(state, action) {
    switch(action.type) {
        case ‘ADD_TODO’:
            const todo = action.payload;
            const todos = [...state.todos, todo];
            return {...state, todos};
        default:
            return state;
    }
}
```

```
state = reducer(state, action)
```

4) Implémentation avec NGRX

Structure des dossiers

- Rappel:
  - Découpe en feature (un module par feature)
  - Découpe par type de fichier (containers/components/services…)
- Pour le store:
  - Nous aurons un dossier « store » à la racine de la feature
    - Dans ce store il y aura une découpe par type de fichier (actions/reducers…)

```
src
    feature
        containers
        components
        services
        …
        store
            actions
            reducers
            …
    …
```

Implémentation avec NGRX – imports

```
import { StoreModule } from '@ngrx/store';
```

Implémentation avec NGRX – app module

```
@ngModule({
    imports: [
        StoreModule.forRoot(reducers)
    ]
})
export class AppModule {}
```

Implémentation avec NGRX – feature module

```
@ngModule({
    imports: [
        StoreModule.forFeature('feature', reducers)
    ]
})
export class FeatureModule {}
```

Implémentation avec NGRX – actions

```
import { Action } from '@ngrx/store';

export const ADD_TODO = '[Todos] Add Todo';

export class AddTodo implements Action {
    public readonly type = ADD_TODO,
    constructor(public payload: Todo) {}
}

...

export type TodosAction = AddTodo | RemoveTodo | ...
```

Implémentation avec NGRX – actions

```
addTodo (event: Todo) {
    this.store.dispatch(new fromStore.AddTodo(event));
}
```

Implémentation avec NGRX – modélisation de l'état

```
export interface TodosState {
    loader: boolean,
    loading: boolean,
    todos: Todo[]
}

const initialState: TodosState = {
    loaded: false,
    loading: false,
    todos: null
}
```

Implémentation avec NGRX – reducers

```
export function reducer (
    state = initialState,
    action: fromTodos.TodosAction
): TodosState {
    switch(action.type) {
        case fromTodos.ADD_TODO:
            const todo = action.payload;
            const todos = [...state.todos, todo];
            return {...state, todos};
    }
    return state;
}
```

Implémentation avec NGRX – préparation du selector

```
export const getTodos = (state: TodosState) => state.todos;
```

Implémentation avec NGRX – enregistrement des reducers

```
import { ActionReducerMap } from '@ngrx/store';

import * as fromTodos from './todos.reducer';

export interface FeatureState {
    todos: fromTodos.TodosState
}

export const reducers: ActionReducerMap<FeatureState> = {
    todos: fromTodos.reducer;
};
```

Implémentation avec NGRX – création d'un selector

```
import { createSelector, createFeatureSelector } from '@ngrx/store';

import * as fromTodos from './todos.reducer';

export interface AppState {
    feature: FeatureState
}

export const getFeatureState = createFeatureSelector<FeatureState>('feature');

export const getFeatureStateTodos = createSelecror(
    getFeatureState,
    fromTodos.getTodos
);
```

Implémentation avec NGRX – utilisation du selector

```
export class TodoComponent implements OnInit {
    todos$: Observable<Todo[]>;

    constructor(private store: Store<fromStore.TodoState>) {}

    ngOnInit() {
        this.todos$ = this.store.select(fromStore.getTodos);
    }
}
```

Implémentation avec NGRX – utilisation du selector

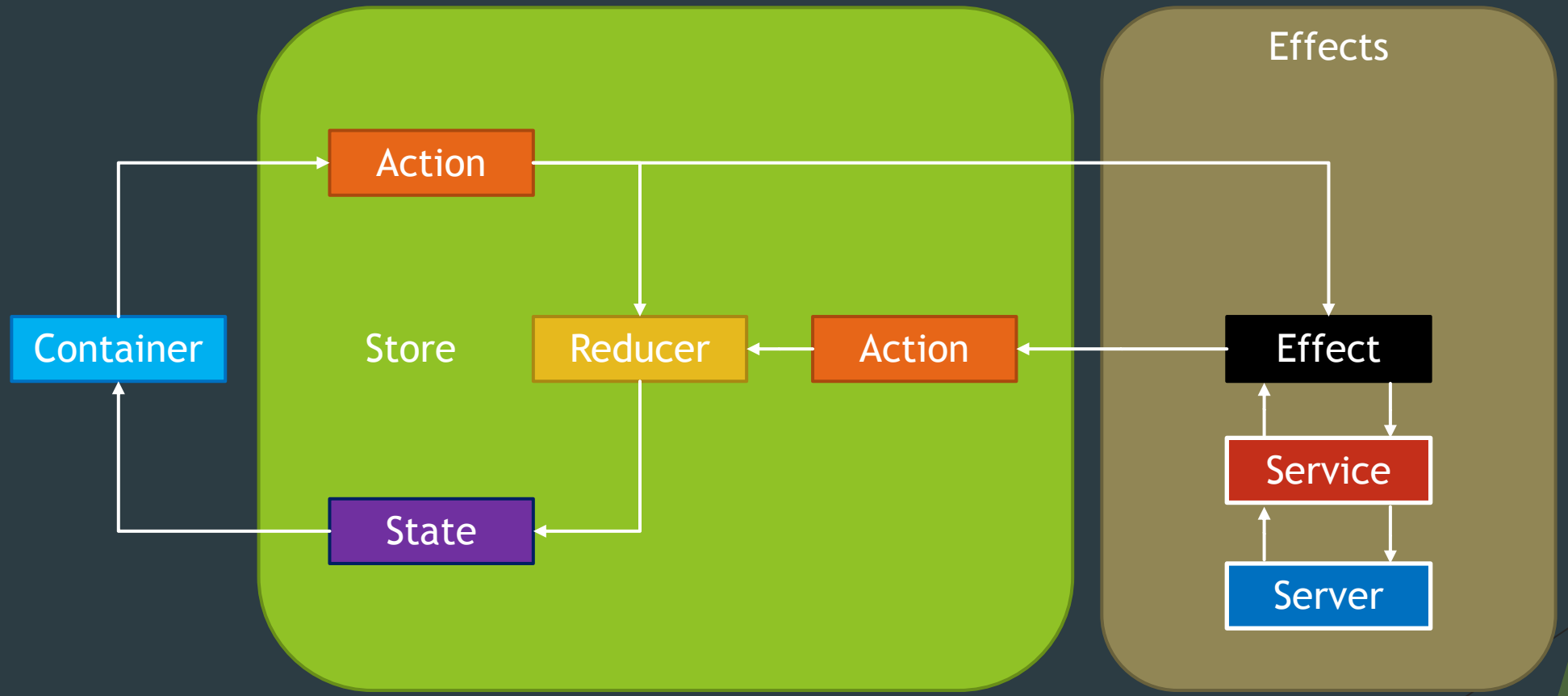
```
<div *ngFor="let todo of (todos$ | async)">
   {{ todo.label }}
</div>
```

5) Gestion des side effects

Gestion des side effects

- Qu’est-ce que la gestion d’effets dans NGRX ?
  - Écoute les actions et réagis autrement qu’en modifiant l’état (c’est le rôle des reducers ça)
  - Des réactions agnostiques à tout composant
  - Peu communiquer avec des webservices

Gestion des side effects



Gestion des side effects

- Exemple :
  - Le composant lance une action « login » (la réaction doit être la même, peu importe le composant)
  - Un effet intercepte l'action et appel le web-service de connexion
  - Une fois que le service a répondu, l'effet lance une autre action « loginSuccess »
  - Cette dernière action est interceptée par un reducer qui met à jour l'état de l'application

Gestion des side effects – imports

```
import { EffectsModule } from '@ngrx/effects';
```

Gestion des side effects – app module

```
@ngModule({
    imports: [
        EffectsModule.forRoot(effects)
    ]
})
export class AppModule {}
```

Gestion des side effects

```
@Effect()
createTodo$ = this.actions$
    .ofType(todoActions.CREATE_TODO)
    .pipe(
        map((action: todoActions.CreateTodo) => action.payload),
        exhaustMap((todo) => {
            return this.todosService
                .createTodo(todo)
                    .pipe(
                        map(todo => new todoActions.LoadTodoSuccess(todo)),
                        catchError(error => of(new todoActions.LoadTodoFail(error)))
                    )
        })
    );
```

Gestion des side effects

```
@Effect({ dispatch: false })
createTodoSuccess$ = this.actions$
    .ofType(todoActions.CREATE_TODO_SUCCESS)
    .pipe(
        map((action: todoActions.CreateTodoSuccess) => action.payload),
        tap(todo => this.router.navigate([`/todos/${todo.id}`]))
    );
```

Références

- https://github.com/ngrx/platform
- https://toddmotto.com/angular/